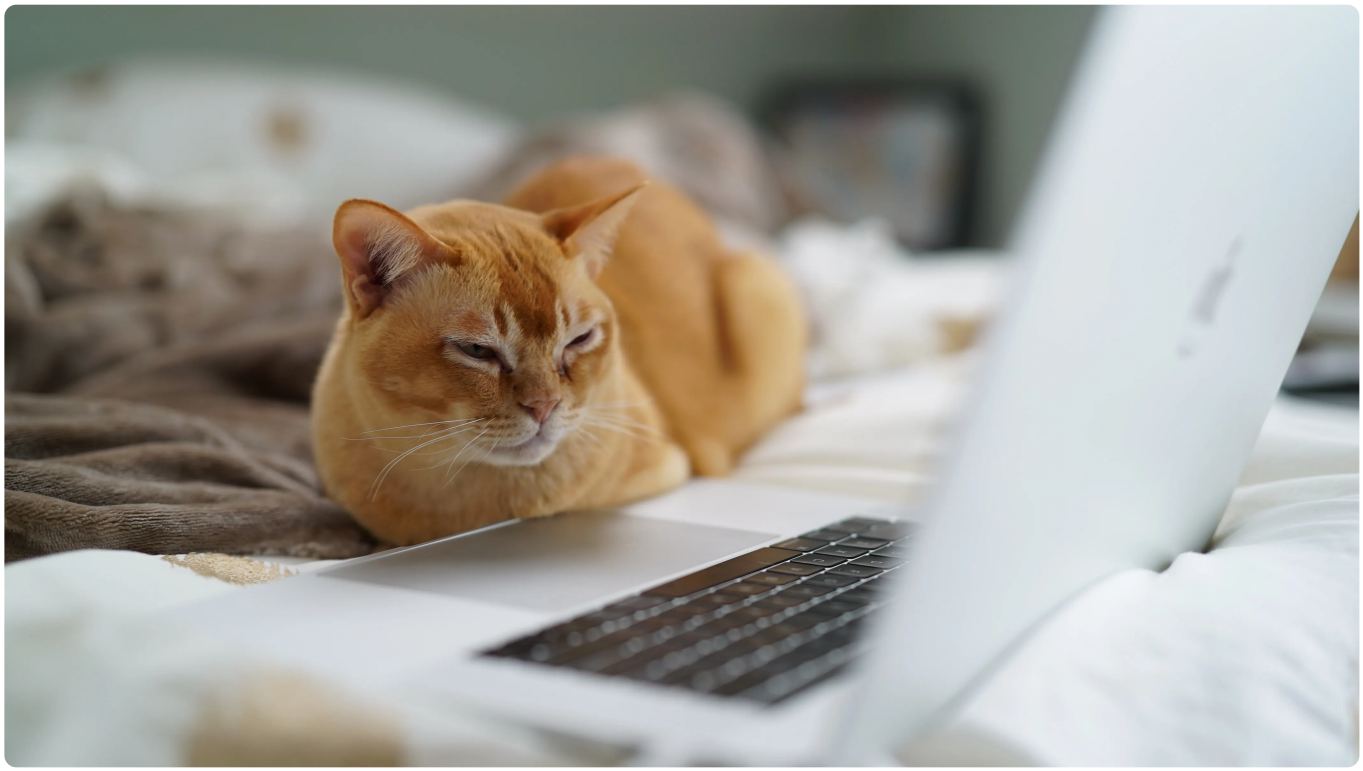
06｜踏出新手村便遭遇大Boss：如何架构低代码的引擎？

2022-03-25 陈旭

《说透低代码》 课程介绍 >



讲述：陈旭
时长 19:50　大小 18.17M

你好，我是陈旭。

可视化开发是所有低代码工具 / 平台（下文简称低代码或 Low Code）的标配，是成为低代码工具 / 平台的一个必要条件。而承载可视化开发的核心基础设施，就是所见即所得的编辑器。

这个编辑器非常重要，它的使用体验、能力和易用性在极大程度上决定了低代码整体的成败。由于编辑器的实现是一个非常大的话题，我们需要分成 8 讲才能说清楚。所以，今天这节课我们先从编辑器的引擎切入，从架构的角度聊一聊应用代码生成器与编辑器之间的关系。

在讨论低代码到底是银弹还是行业毒瘤那一讲中，我们了解到低代码的开发过程就是不停地在描述和细化一个业务最终的样子。这就要求低代码编辑器能实时反馈出开发人员的操作结果，这也就是所见即所得的含义。

而通过模拟，是难以保持“所见”与“所得”的一致性的。为了达到“所见”与“所得”的高度一致，我们就需要实时地把应用创建出来，创建应用的过程就是生成并运行应用代码的过程。显然，生成应用代码是这一切的基础。编辑器在收集到开发人员的操作后，应该立即生成应用的代码。

那么，生成代码的功能在架构上和编辑器可以有什么样的关系呢？不同的关系对低代码长期演进会有什么样的影响呢？不同的组织应该如何选择合适的架构呢？这正是你我今天要探讨的主要内容。

代码生成器与编辑器的关系

我了解过许多失败的案例，它们多数有一个共同特点：总是开始于一个玩具。故事基本上可以归纳为某个人或者团队因为偶然的心血来潮，写了个小玩意，有个界面，支持拖来拖去，可以生成特定功能。老板了解后都觉得有用（老板觉得没用的那些自然就不会被了解到了），于是加大投入，想做得更大、更好。结果，初创团队“鸡血”了几个月后，基本就玩不转了，要么销声匿迹，要么推倒重来。

其实界面拖拖拽拽生成特定功能的门槛并不高，但是要承载厚重的低代码战车，则需要有很深远的设计和思考，其中最重要的一环是如何生成应用的代码。

代码生成器与编辑器之间的关系，可以大致分为这几个层次：

Level 1：没有代码生成器的概念，或者极其粗糙；

Level 2：有相对独立的模块用于生成代码，但该模块与编辑器耦合严重；

Level 3：代码生成器与编辑器基本相互独立，具有同等地位；

Level 4：插件系统与生态，编译器必须再次抽象才能实现插件系统。

Level 1

如果连代码生成器的概念都没有，或者由散落在代码仓库各个角落里的三两个函数构成，那么这样的编辑器显然是无法区分编辑态和运行态的。即使有代码生成器，也是在编辑器的功能基础上通过 if else 来实现的，比如引入一个只读状态，在这个状态下无法编辑，并将它作为运行态来用。很显然这样的实现方式，是极不妥的。由于没有足够的抽象，功能点加多了后，编辑器的代码就会变得极其难以维护。

关于代码的可维护性，我常常说的一句话是：**一个 if 一个坑**。使用 if 就等于在尝试对事物状态进行枚举。简单事物可以枚举所有状态，但多数事物的状态是不可枚举的。每少考虑一个状态就是一个 bug 或需求，同时也是对已有逻辑的一次冲击。而为了能够区分出相似的情况，往往需要增加新的 flag，在 flag 数量达到某个数之后，这份代码就再也改不动了。

Level 2

当你意识到 if 解决不了问题时，往往就会开始考虑对编辑器做抽象了。而编辑态和运行态是编辑器两种最基本的状态，这两种状态有很多共同的部分，也有一小部分差异。所以抽象的第一步就是把公共部分抽取出来，并在编辑态和运行态下改写公共层里的某些行为。这是 OOP 思想的最基础的应用。

我们这一讲先不展开讲解如何去抽象，但是有一点是非常明确的：生成代码的能力是两个状态都需要的，应该归入公共层中。所以，很自然的做法是把散落在代码仓库各角落里的那三两个函数挪到一起，最起码是放到同一个文件里去，然后适当调整入参和依赖，让它们能恢复正常功能。**更进一步，需要将原有的 if else 改用 OOP 的多态特性来实现**。

到了这个时候，你就需要好好考虑一下 TypeScript 了。在这节课中，我不想展开讨论是 TypeScript 好，还是 JavaScript 好这样的细节，但是**我的建议是此时应该坚决引入 TypeScript**。因为 TypeScript 提供了一套非常完整的 OOP 实现，而 OOP 是用于对复杂事物做抽象的必备武器，基于 JavaScript 原型链自行实现的 OOP 没有那么完善（我敢保证！），弃之勿留恋！

另一个关键原因是，TypeScript 提供了极完善的静态类型支持。记住，你正在开发一个编辑器，除非你打算一直把它当作一个玩具，否则你要先做好至少 5~10 万行代码的觉悟。这样量级下的代码，如果没有类型的辅助，我们的开发效率将是有静态类型支持下的 1/2~1/3，甚至更低。

当你的编辑器的公共层里有了一个相对独立的代码生成器模块之后，编辑器的长期演进就有了一个架构基础，但也仅此而已。

此时你的 APP 依然只能在编辑器上运行，无法独自运行，这会导致将来你的业务应用和你的低代码平台之间产生耦合。要么你把应用包住对外提供业务价值，要么应用把你包住，这取决于谁更强势。但无论如何，耦合是不可避免的。

到了这里，**你就需要好好思考一下将来你和你的应用团队之间的关系了**。如果你的老板本就打算或允许你们耦合在一起，那么到这里你就可以继续开发编辑器的其他功能了。反之，这事还没完。

提醒一点：请务必确认你的老板知晓平台和应用是可以实现解耦的，否则请适当给予他一些提示（写一份 PPT 吧，他会因此更喜欢你的），并且确认他确实经过一番思虑后才做出保持耦合的决策。

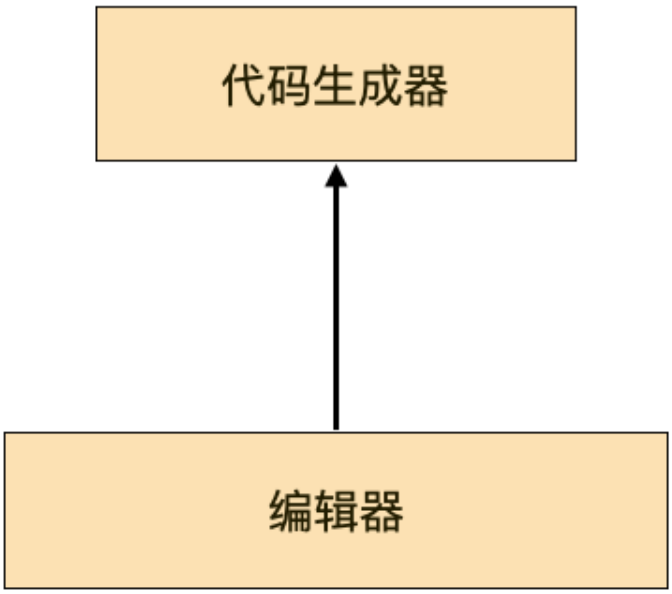
因为，与应用之间保持耦合这个事情，开弓就没有回头箭了。如果你的老板依然保持模棱两可的状态，那么我建议你将生成代码这事继续完善下去，避免日后他后悔了并给你下一个你压根就无法实现的任务。与应用解耦状态下是可以融合部署对外提供业务的，反之是走不通的。

Level 3

为了解除平台与应用之间的耦合关系，我们需要将代码生成器进一步独立出来，提到与编辑器同等地位上。在 Level 2 中，我们是把它当做一个公共模块（一个库），而到 Level 3，我们是要把它作为与编辑器对等的一个独立功能来实现。

一个比较好的实现方式是将它做成一个命令行，可以在 shell 终端里跑。命令行的好处很多，主要很容易与 DevOps 流水线结合使用，或者实现各种自动化。应用跑一下命令行就可以生成一份可以独立运行的代码出来，这样你爱放哪运行都与编辑器无关了。

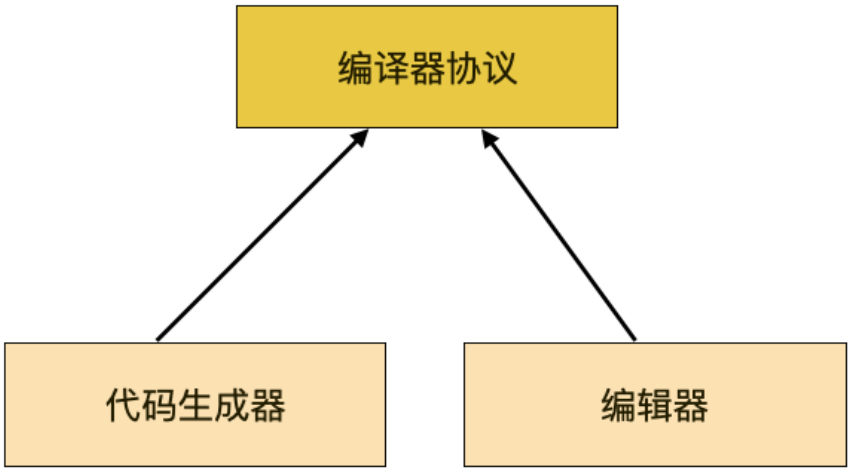
将代码生成器与编辑器彻底分离，让它们可以脱离对方独立运行，需要对代码做更深一层次的抽象。在 Level2 中，两者的关系是下面这样子的：



两者之间是紧耦合的关系，这意味着代码生成器很难被扩展和定制。不可否认的是，业务的发展速度是远大于代码生成器的，因此业务的需求永不终结。在代码生成器难以扩展和定制的前提下，低代码平台非常容易为了满足某个紧急业务需求，不得不将该业务耦合到代码生成器的实现中。

一旦开了这样的口子，代码生成器架构的腐化也就开始了，逐渐就会更越来越多的业务耦合进来，不需要太多时间，这样的代码生成器就无法继续演进下去了。

而在 Level 3 架构下，编辑器和代码生成器之间的关系变成了下面这样：

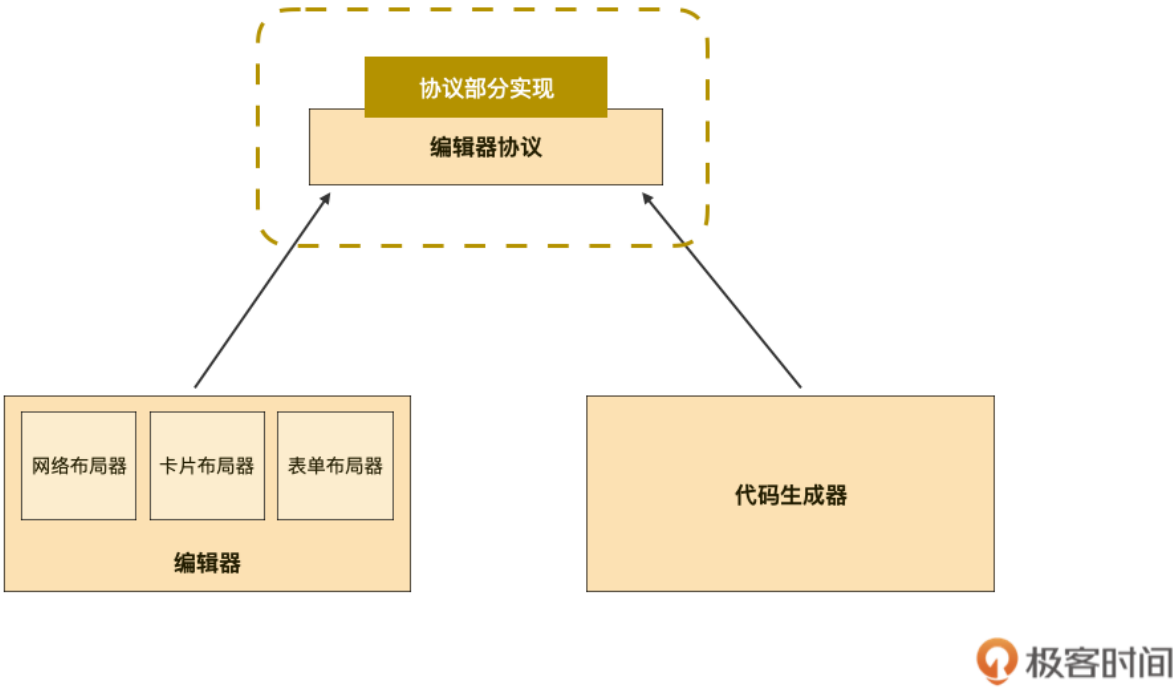


极客时间

代码生成器和编辑器处于同一层级，并且**引入了协议层**。协议层的作用之一，就是对生成代码各种功能的 API 做出抽象和约束。代码生成器主要职责是实现这些协议，而编辑器的主要职责则是调用协议层提供的 API 来完成代码生成等一系列上层活动。

然而，**引入协议层的更深层目的，是为了方便应用扩展和定制**。从实现角度看，协议层实际上就是一堆接口（interface），这些接口的默认实现是由代码生成器按照其通用的、与应用无耦合的方式来实现的。当通用的实现不满足应用的需要时，应用要有渠道来编写适合其自身的协议层的实现，并覆盖掉默认通用的实现，从而实现编译器的定制化和按需扩展 *。

当然，让应用实现一套完整的编译协议是不合理的。我们可以在编译协议层里加入部分实现，这样应用在定制时就可以只覆盖少数不适合的实现，复用大部分的已有实现，这样可以大大减轻应用定制的工作量和难度。因此，这个架构的实际关系是下面这样的：



虚线框里的部分，实际上就是应用定制代码生成过程所需的 SDK。这就是 Level 3 的架构，它不仅具有更好的长期演进基础，还具有非常好的、非常优雅的扩展性和定制性。至此，我们可以给代码生成器换一个高大上的名字了，将其称为**编译器**。

Level 3 只提供了扩展和定制的可能。如果我们对这些扩展和定制点再做一番系统性的设计，在此基础上抽象出编译流程的各个时机（可称为编译生命周期），以及对各个时机具备的可扩展点加以规范，从而形成一套完备的插件开发能力，那我们可以让这个架构演进到 Level 4 了。**Level 4 的最主要目的要形成完善的插件系统，进而具备打造生态圈的基础**。Level 4 已经超出这节课的范畴太多了，在这个专栏的最后，我会留一讲专门来说这个话题。

最后，我再分享一点我负责的低代码平台目前的架构演进的状况，可以作为案例给你一点参考。

其实，我也不是一开始就有如此清晰的分层演进的考虑的。不过庆幸的是，我在编码开始前多花了点时间做了类似前文的思考，所以我直接跳过了 Level 1 和 Level 2，直接按照 Level3 的架构目标开始编码。但为了更快做出 MVP（最小可用版本），我并没有严格按照 Level 3 的架构实现，而是偷了点巧，先把编译器当做一个库实现出来，在 MVP 完成之后老板认可了，才专门花了点时间把编译器独立出来，完成 Level 3 架构的转型。

我想说的是，这里所说的架构分层是一个目标，你在实现时，可以根据实际情况（工期、人力等）灵活调整。但即使我未严格遵循 Level3 的架构要求实现，我还是让编译器尽可能地与编辑器保持独立，同时在过程中逐渐抽取 API 作为协议层。

目前，我负责的低代码平台正处于 Level 3.5，它已经拥有了相对完善的插件系统，具有发展生态的基础了。演进到了这个阶段，它的编译器已经很稳定了，目前它的主要任务是推广和布道，枯燥且乏味。

低代码编辑器要能实现最基础的“所见即所得”的闭环（就是达到给老版演示的最低要求），就不得不实现生成代码的功能。这个看似简单的小功能，背后却是牵动着日后长期演进的许多考虑。

至此，你是否觉得我们这节课的标题起得非常好？低代码开发者刚走出新手村碰到的第一个怪物就是一个实力 Boss，关键是这个 Boss 看起来和一个普通小怪差不多，我们冲上去居然不会被它秒杀，嗑点药居然还能站得住。

当你磨了半天都没能杀掉它的时候，不妨停下来仔细观察它，你会发现原本你以为的青铜怪居然是一位王者。此时我们应该立刻回城并想办法升级（氪金 ^_^）装备和技能，决不可恋战！

生成代码总体流程

接下来，我再说说实现编译器的两种可行方法、及其优劣和适用场景。注意，下面讨论的默认应用是以 UI 复杂且交互密集为主要特征的 PC 端网页，具有类似性质的移动端页面也适用。C/S 架构下的 UI 不在讨论范围内。

编译器的输入是一组结构化的数据。即使编辑器与编译器之间采用某种 DSL（领域编程语言）作为协议，但是输入给编译器之前，我们肯定要将输入数据解析为结构化的数据。为了描述方便，我使用 SVD 这个内部术语来指代这组结构化数据。编译器的唯一任务就是将 SVD 转成代码。这里有两种编译选择：

直接法：直接将 SVD 生成出浏览器能识别的代码；

间接法：先将 SVD 生成出某种 MVVM 框架的代码，再利用其编译器进一步编译成浏览器能识别的代码。

两种方式各有优劣，直接法的最大好处是架构简单，特别是无需再引入和协调另一个编译器，能少写不少代码；其次是效率高，JiT 编译器只需间接法的约 20%~30%，几乎可以做到全程百毫秒以内的 JiT 编译消耗。

当然，直接法的代价也是非常明显的。它最大的问题在于，你必须考虑你所在场景里的前端技术栈选型。因为到了 Level 4 的时候，你就需要考虑生态的问题了，而现在前端生态最大的割裂莫过于 AVR（Angular、VUE、React）等框架的技术栈选型了，极少有裸奔（含 jQuery）的。如果你期望到时能和它们玩到一起去，那么现在就应该选择相同的技术栈。

无论你选择了 AVR 中的哪个，都意味着你只能采用间接法。如果你处于一个强势的机构并且你老板坚定为你站台，所有不用你平台的应用团队一律给考核 C，那么可以无视这条（你好幸福）。或者你的平台不打算给内部开发人员使用，也可以不考虑这条。

直接法的另一个约束在于页面组件集的实现。等等！编译器和组件集居然也能扯上关系？实际上，我认为低代码实现之初主要的基础设施之一就是要有一套合适的、可控的组件集。你一定要记住，细节管控的程度会对低代码的易用性和适用性产生决定性的影响，**而权衡之下最合适的细节管控粒度就是 Web 组件集。**

Web 组件集将大量的 HTML/CSS 细节封装掉，只通过 API 的方式暴露出来，而低代码通过引入一套组件集可以屏蔽掉 HTML/CSS 的所有细节，大大提升易用性。如果你的组件集是用 jQuery 做的，那么可采用直接法生成代码。但是，我猜大概率你不会这么做，即使曾经有 jQuery 实现的组件集，也早就被你嫌弃并使用 AVR 之一重写了（这是一个重复再造轮子并获得晋升的绝佳机会，如果你没抓住，那就太可惜了）。

如果你选用的组件集是 AVR 之一实现的，那很可惜，你只能采用间接法。不过，随着 Web Component 的发展，目前 AVR 都已推出 Web Component 的转译器。利用 Web Component 技术，你就可以直接使用浏览器原生技术来动态渲染 APP。如果你有条件使用这个技术，那也可以无视这条约束了。比如，华为云的低代码平台就是采用这个技术来屏蔽 AVR 差异的，这是一个成功的案例。莫春辉老师在GMTC2021 深圳站上详细介绍了这个案例，感兴趣可以去看看。

只要符合上述两个条件之一，就只能采用间接法。现在我再说说间接法。

估计你已经看出来了，间接法就是一个备胎。只有在你没有条件使用直接法的时候，才用它。它的好处是能兼容 AVR 等各种框架，对 Web 组件集的要求也会低很多，也不需要考虑组件集提供 Web Component 版的工作放在哪。

间接法的代价是架构复杂，需要额外协调 AVR 提供的编译器，这事需要额外消耗许多脑细胞，而且社区里提供的资料都是如何“正常”地去用它。而协调这些编译器这事却是不走寻常路，这样碰到坑大概率要去看编译器的源码，难度很大。当然，如果你是一名技术极客，把这事当做一个优势看待，我也不拦你。

其实，间接法有一个绝对优势，那就是可以支持 Low Code 和 Pro Code 混合开发一个 APP。这是直接法所做不到的。支持混合开发的重要意义在于，你可以相对妥善地处理好存量代码（没错，就是你的低代码做出来之前的那些页面代码）。这点有机会我们再聊。

最后，如果你和我们一样选了 Angular，那恭喜，你还需要额外多协调 TypeScript 编译器。我在 QCon+ 有一次在线分享，讲的就是如何在浏览器构建 TS 的 JiT 编译器以打通整个编译流水线，那个分享具有非常好的实操价值，你可以去看看 PPT（提取码：lcdp）。

总结

今天，我们从架构的角度，对低代码编辑器与应用代码生成器（编译器）之间的关系做了详细的讨论。根据抽象程度的不同，我把代码生成器划分为 4 个层级，并详细说明了前三个层级的特点和适用的场合，以及各个层级对低代码平台的长期演进会产生啥样的影响：

Level 1：基本上就是一个玩具，适合用作试验品，用于快速试错、收集你的低代码潜在用户的反馈，如果确定要开发一个可以称为工具的低代码，那么强烈建议推到重来；

Level 2：基本上可以称为是一个低代码工具了，应该可以一定程度上发挥低代码的优势来，也具备了长期演进的架构基础，但是扩展性和定制性也很弱，容易与应用产生耦合，因此只能在小范围内使用，无法规模推广；

Level 3：基本上可以称为是一个低代码平台了，它的架构清晰、层次分明，具有非常好的长期演进能力，且具备良好的扩展性和定制能力，能处理好业务团队提的各种需求，对于其能力之外的需求，也可以通过扩展和定制的方式优雅地处理，适合大规模推广。

同时，这一讲我们也讨论了实现代码生成器的不同方法，主要侧重讨论了如何根据现有技术选型、长期演进方面挑选合适的实现线路。

直接法架构简单、性能好、实现工作量低，但是选用的先决条件苛刻，主要有两点：一是你所在机构的前端技术选型；二是所用组件集的前端技术选型。间接法作为直接法的备胎，当没有条件使用直接法时，只能选择间接法。间接法架构复杂、JiT 及实时渲染性能差、需要额外协调所用框架的编译器等。但间接法额外带给我们一个直接法所没有的能力：允许 Low Code 与 Pro Code 混合使用，进而可以相对妥善地解决低代码模式与传统模式存量代码之间的关系。

在下一讲中，我将会从实操角度，介绍如何做出一个和能人工一样实现几乎任何业务功能的代码生成器，以及一个 Level3 层级的编译器如何实现。

思考题

根据抽象程度的不同，应用代码生成器与编辑器之间可以分为几个层级？各个层级的关键特征是什么？不同层级对低代码平台长期演进具有什么样的意义？欢迎在留言区写下你的看法。

我是陈旭，我们下节课见。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 05 | 基础设施 ：启动低代码平台研发之前，你需要有什么家底？

精选留言 (1)

写留言

huangshan

2022-03-26

老师，有几个问题需要请教一下 1、编辑器中组件和组件之间的事件流如何控制，比如A组件的事件会影响B组件的刷新，是在编辑器页面配置，还是通过写代码的方式 2、低代码的解析引擎，在需要手动开发时，如何做到二次扩展开发。 3、低代码在某些场景下可以做到零代码吗？如果可以转到零代码开发，是不是受众不一定非得是程序员，比如运维人员就可以做到页面开发。

展开

作者回复: 问题1：我们是难过一个全局的事件中心来收发事件的，一个典型的订阅者模式场景。通过事件广播可以解决你在问题1里举例的场景，此外我们还可以通过数据绑定来自动完成这种简单的情形，当A的属性发生变化，绑定了A的其它所有组件可以直接收到通知并执行预设动作。
问题2：这个问题比较复杂，我在后续几讲里会涉及到，并在常规更新阶段的最后一讲会专门总结和扩展，欢迎继续关注和学习。
问题3：某些场景做到零代码是当然可以的，比如我们现在正在把大屏做到零代码给乙方用户直接使用，这些人是只有业务能力没有编程能力。另外，表单场景我们已经基本做到零代码了